

# **MOBPRO - Mobile Programming**

## **Zusammenfassung FS 2019**

Maurin D. Thalmann

11. März 2019

# Inhaltsverzeichnis

<b>1</b>	<b>Android 1 - Grundlagen</b>	<b>3</b>
1.1	Komponenten	3
1.2	Das Android-Manifest	4
1.3	Activities & Aufruf mit Intents	4
1.3.1	Beispielaufruf Expliziter Intent	5
1.3.2	Beispielaufruf Impliziter Intent	6
1.4	Activities & Subactivities	6
1.5	Lebenszyklus & Zustände von Applikationen/Activities	6
1.5.1	Lifecycle einer Applikation	7
1.6	Charakterisierung einer Activity	8
1.6.1	Zustandsänderung - Hook-Methoden	8
1.7	Android - Hinter den Kulissen	9
1.7.1	Android-Security-Konzept	10
<b>2</b>	<b>Android 2 - Benutzerschnittstellen</b>	<b>11</b>
2.1	GUI einer Activity	11
2.2	XML-Layout	12
2.2.1	Constraint-Layout	12
2.2.2	LinearLayout	12
2.3	Ressourcen, Konfigurationen und Internationalisierung	13
2.4	UI-Event-Handling	14
2.4.1	GUI-Events	14
2.4.2	Exkurs: Data Binding	15
2.5	Options-Menü	16
2.6	Adapter-Views	17
2.6.1	AdapterViews & ListActivity	17
2.6.2	android.widget.Spinner	18
2.6.3	android.widget.ListView	19
2.6.4	android.app.ListActivity	19
2.7	ViewModel - Konfigurationswechsel & temporäre Datenspeicherung	20
2.8	Rückmeldungen an den Benutzer	21
2.8.1	Toast	21
2.8.2	Alert-Dialog	21
2.8.3	Notifications (Status-Bar)	24
<b>3</b>	<b>Android 3 - Persistenz &amp; Content Providers</b>	<b>25</b>
3.1	(Shared) Preferences	25
3.1.1	Darstellung User-Preferences	25
3.1.2	PreferenceFragment	27
3.1.3	Default-Präferenzen	27
3.2	Dateisystem	28
3.2.1	Exkurs: Permission-Model	28
3.2.2	Exkurs ff: Runtime Permissions	29
3.2.3	Exkurs ff: Persistenz mit Datei	29
3.3	Datenbank (Room)	30
3.3.1	Room - Code-Beispiele	31
3.3.2	Room - Daten mit Entitäten definieren	32
3.3.3	Room - Beziehungen modellieren	32
3.4	Mit DAOs auf Daten zugreifen	33
3.4.1	Convenience Queries	33
3.4.2	Custom Queries mit @Query	34
3.5	DB-Einträge in einer Liste anzeigen	35
3.5.1	Room - weitere Themen	35
3.6	Content Providers	36
3.6.1	Standard Content Providers	36

3.7	Exkurs : REST-ful Webservices . . . . .	36
3.8	Content Resolver & Content Provider . . . . .	37
3.8.1	Zugriff auf Daten über Content Resolver & Query . . . . .	37
3.9	Eigener Content Provider . . . . .	38

# 1 Android 1 - Grundlagen

Informationen zur Androidprogrammierung können stets dem Android Developer Guide entnommen werden unter: *developer.android.com* Apps sollen grundsätzlich gegen das aktuellste API entwickelt werden, aktuell API Level 28 Android 9 „Pie“. Im Gradle-Build-Skript werden deshalb folgende SDK-Versionen festgehalten:

**minSdkVersion** Mindestanforderung an die SDK, Minimum-Version

**targetSdkVersion** Ziel-SDK-Version, auf welcher die App lauffähig sein soll

**compileSdkVersion** Version mit welcher die App (APK) erstellt wird, meist gleich der Target-Version

**ART (Android Runtime)** verwaltet Applikationen bzw. deren einzelne Komponenten:

- Komponente kann andere Komponente mit Intent-Mechanismus aufrufen
- Komponenten müssen beim System registriert werden (teilweise mit Rechten = Privileges)
- System verwaltet Lebenszyklus von Komponenten: Gestartet, Pausiert, Aktiv, Gestoppt, etc.

## 1.1 Komponenten

Applikationen sind aus Komponenten aufgebaut, die App verwendet dabei eigene Komponenten (min. eine) oder Komponenten von anderen, existierenden Applikationen.

Name	Beschreibung
<b>Activity</b>	UI-Komponente, entspricht typischerweise einem Bildschirm
<b>Service</b>	Komponente ohne UI, Dienst läuft typischerweise im Hintergrund
<b>Broadcast Receiver</b>	Event-Handler, welche auf App-interne oder systemweite Broadcast-Nachrichten reagieren
<b>Content Provider</b>	Komponente, welche Datenaustausch zwischen versch. Applikationen ermöglicht

**Activity** entspricht einem Bildschirm, stellt UI-Widgets dar, reagiert auf Benutzer-Eingabe & -Ereignisse. Eine App besteht meist aus mehreren Activities / Bildschirmen, die auf einem „Stack“ liegen.

Basisklasse: *android.app.Activity*

**Service** läuft typischerweise im Hintergrund für unbeschränkte Zeit, hat keine graphische Benutzerschnittstelle (UI), ein UI für ein Service wird immer von einer Activity dargestellt.

Basisklasse: *android.app.Service*

**Broadcast Receiver** ist eine Komponente, welche Broadcast-Nachrichten empfängt und darauf reagiert. Viele Broadcasts stammen vom System (Neue Zeitzone, Akku fast leer,...), App kann aber auch interne Broadcasts versenden.

Basisklasse: *android.content.BroadcastReceiver*

**Content Provider** ist die einzige direkte Möglichkeit zum Datenaustausch zwischen Android-Apps. Bieten Standard-API für Suchen, Löschen, Aktualisieren und Einfügen von Daten.

Basisklasse: *android.content.ContentProvider*

## 1.2 Das Android-Manifest

**AndroidManifest.xml** dient dazu, alle Komponenten einer Applikation dem System bekannt zu geben. Es enthält Informationen über Komponenten der Applikation, statische Rechte (Privileges), Liste mit Erlaubnissen (Permissions), ggf. Einschränkungen für Aufrufe (Intent-Filter). Es beschreibt die statischen Eigenschaften einer Applikation, beispielsweise:

*(Diese Infos werden bei der App-Installation im System registriert, zusätzliche Infos (Version, ID, etc.) befinden sich im Gradle-Build-Skript (können build-abhängig sein))*

- Java-Package-Name
- Benötigte Rechte (Internet, Kontakte, usw.)
- Deklaration der Komponenten
  - Activities, Services, Broadcast Receivers, Content Providers
  - Name (+ Basis-Package = Java Klasse)
  - Anforderungen für Aufruf (Intent) für A, S, BR
  - Format der gelieferten Daten für CP

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ch.hslu.mobpro.hellohslu">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Abbildung 1: Beispiel eines Android-Manifests

## 1.3 Activities & Aufruf mit Intents

Zwischen Komponenten herrscht das Prinzip der losen Kopplung:

- Komponenten rufen andere Komponenten über Intents (= Nachrichten) auf
- Offene Kommunikation: Sender weiss nicht ob Empfänger existiert
- Parameterübergabe als Strings (untypisiert)
- Parameter: von Empfänger geprüft, geparkt & interpretiert (oder ignoriert)
- Keine expliziten Abhängigkeiten → Robuste Systemarchitektur

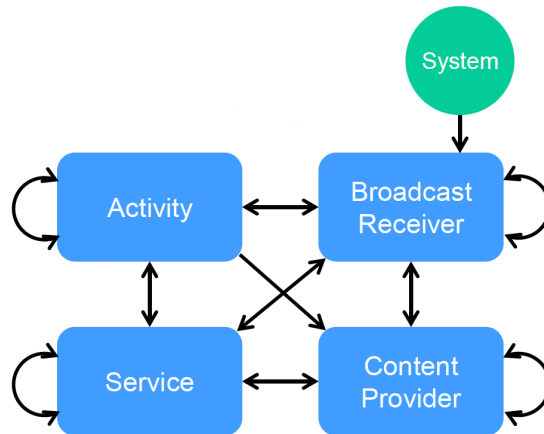


Abbildung 2: Kommunikation zwischen Komponenten mit Intents

Intents werden benutzt, um Komponenten zu benachrichtigen oder um Kontrolle zu übergeben. Es gibt folgende zwei Arten von Intents:

**Explizite Intents** adressieren eine Komponente direkt

**Implizite Intents** beschreiben einen geeigneten Empfänger

**WICHTIG:** Activities müssen immer im Manifest deklariert werden, da sie sonst nicht als „public“ gelten und eine Exception schmeissen. Das geht auch ganz einfach folgendermassen im Manifest unter „application“:

---

```

1 <activity android:name=".Sender" />
2 <activity android:name=".Receiver" />

```

---

### 1.3.1 Beispielaufruf Expliziter Intent

#### Sender Activity:

---

```

1 public void onClickSendBtn(final View btn) {
2     Intent intent = new Intent(this, Receiver.class);
3     // Receiver.class ist hier der explizite Empfaenger
4     intent.putExtra("msg", "Hello World!");
5     startActivity(intent);
6 }

```

---

#### Receiver Activity:

---

```

1 public void onCreate(Bundle savedInstanceState) {
2     // ...
3     Intent intent = getIntent();
4     String msg = intent.getExtras().getString("msg");
5     displayMessage(msg);
6 }

```

---

### 1.3.2 Beispielaufruf Impliziter Intent

#### Sender Activity:

```
1 Intent browserCall = new Intent();
2 browserCall.setAction(Intent.ACTION_VIEW);
3 browserCall.setData(Uri.parse("http://www.hslu.ch"));
4 startActivity(browserCall);
```

*ACTION\_VIEW* ist hierbei kein expliziter Empfängertyp, sondern nur eine gewünschte Aktion. Die mitgegebene URL wird auch ein *Call Parameter* genannt. Gesucht ist in diesem Fall eine Komponente, welche eine URL anzeigen/verwenden kann.

## 1.4 Activities & Subactivities

**Activity Back Stack:** Activities liegen aufeinander wie ein Stapel Karten, neuste Activity zuoberst und in der Regel ist nur diese sichtbar (Durch Transparenz sind hier Ausnahmen möglich). Durch „back“ oder „finish“ wird die oberste Karte entfernt und man kehrt zur zweitletzten Activity zurück. Mehrere Instanzen derselben Activity wären mehrere solche Karten, das Verhalten kann jedoch konfiguriert werden (z.Bsp. maximal eine Instanz, mehrere Activities öffnen, etc.)

**(Sub-)Activities und Rückgabewerte:** Eine Activity kann Rückgabewerte einer anderen (Sub-)Activity erhalten.

```
1 // 1. Aufruf der SubActivity mit
2 startActivityForResult(intent, requestId)
3
4 // 2. SubActivity setzt am Ende Resultat mit
5 setResult(resultCode, intent) // intent als Wrapper fuer Rueckgabewerte
6
7 // 3. SubActivity beendet sich mit
8 finish()
9
10 // 4. Nach Beendigung der SubActivity wird folgendes im Aufrufer aufgerufen:
11 onActivityResult(requestId, resultCode, intent)
12 // resultCode: RESULT_OK, RESULT_CANCELLED
```

## 1.5 Lebenszyklus & Zustände von Applikationen/Activities

Das System kann Applikationen bei knappem Speicher ohne Vorwarnung terminieren (nur Activities im Hintergrund, dies geschieht unbemerkt vom User, die App wird bei Zurücknavigation wiederhergestellt). Eine Applikation kann ihren Lebenszyklus demnach nicht kontrollieren und muss in der Lage sein, ihren Zustand speichern und wieder laden zu können. Applikationen durchlaufen mehrere Zustände in ihrem Lebenszyklus, Zustandsübergänge rufen Callback-Methoden auf (welche von uns überschrieben werden können).

#### Activity-Zustände:

Zustand	Beschreibung
Running	Die Activity ist im Vordergrund auf dem Bildschirm (zuoberst auf dem Activity-Stack für die aktuelle Aufgabe).
Paused	Die Activity hat den Fokus verloren, ist aber immer noch sichtbar für den Benutzer.
Stopped	Die Activity ist komplett verdeckt von einer andern Activity. Der Zustand der Activity bleibt jedoch erhalten.

### 1.5.1 Lifecycle einer Applikation

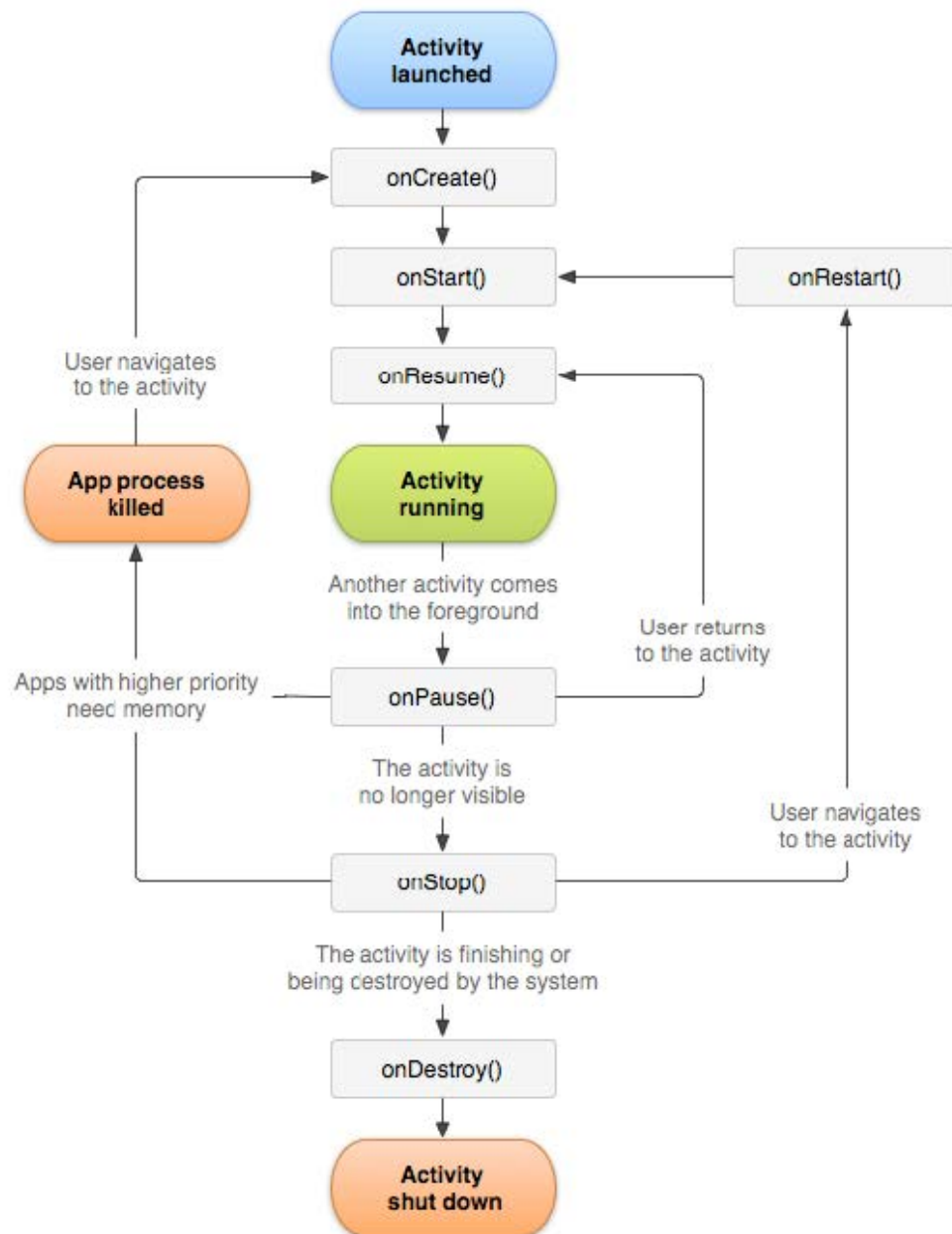


Abbildung 3: Lifecycle einer Applikation

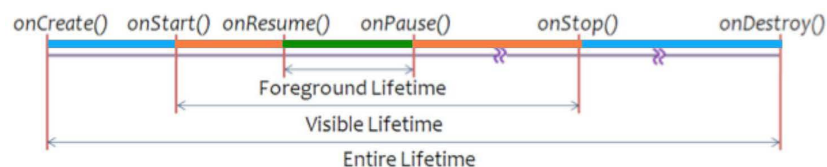


Abbildung 4: Lebenszeiten der einzelnen App-Zustände



## 1.6 Charakterisierung einer Activity

- Muss im Manifest deklariert werden
- GUI-Controller
  - Repräsentiert eine Applikations-/Bildschirmseite
  - Definiert Seitenlayout und GUI-Komponenten
  - Kann aus Fragmenten (= „Sub-Activities“) aufgebaut sein
  - Reagiert auf Benutzereingaben
  - Beinhaltet Applikationslogik für dargestellte Seite

### Beispiel einer Activity:

---

```
1 public class Demo extends Activity {  
2     // Called when the Activity is first created  
3     public void onCreate(Bundle savedInstanceState) {  
4         super.onCreate(savedInstanceState);  
5         setContentView(R.layout.main); // Definiert Layout und UI  
6     }  
7 }
```

---

### 1.6.1 Zustandsänderung - Hook-Methoden

Das System benachrichtigt Activities durch Aufruf einer der folgenden Methoden der Klasse *Activity*:

- void onCreate(Bundle savedInstanceState)
- void onStart() / void onRestart()
- void onResume()
- void onPause() → *bspw. Animation stoppen*
- void onStop()
- void onDestroy() → *bspw. Ressourcen freigeben*

Durch das Überschreiben dieser Methoden können wir uns in den Lebenszyklus einklinken. Immer **super()** aufrufen, sonst wirft es eine Exception.

## 1.7 Android - Hinter den Kulissen

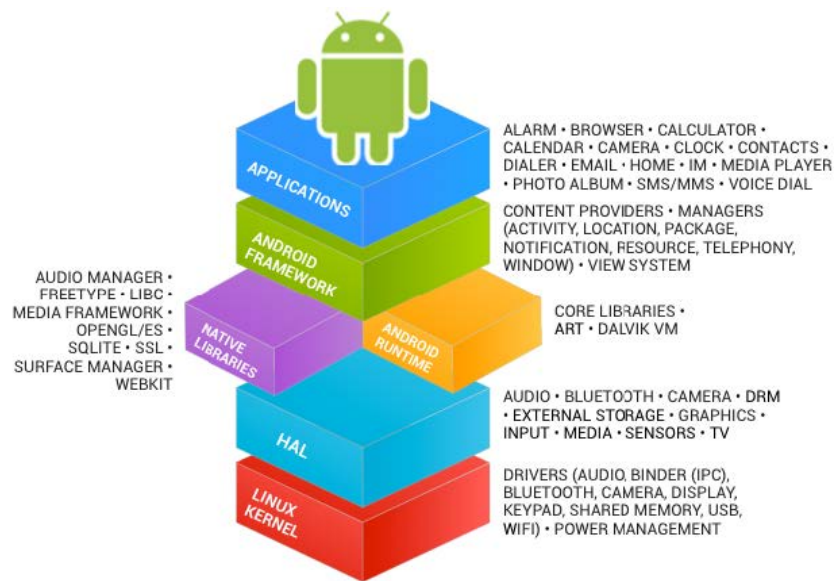


Abbildung 5: Der Android-Stack

- **Linux-Kernel:** OS, FS, Security, Drivers, ...
- **HAL (Hardware Abstraction Layer):** Camera-, Sensor-, ... Abstraktion
- **ART** (Android Runtime)
  - Jede App in eigenem Prozess
  - Optimierte für mehrere JVM auf low-memory Geräten
  - Eigenes Bytecode-Format (Crosscompiling)
  - JIT und AOT Support
- **Native C/C++ Libraries:** Zugriff via Android NDK
- **Android Framework:** Android Java API
- **Applications:** System- und eigene Apps

### 1.7.1 Android-Security-Konzept

#### Sandbox-Konzept:

- Jede laufende Android-Anwendung hat seinen eigenen Prozess, Benutzer, ART-Instanz, Heap und Dateisystembereich → jedes App hat eigenen Linux-User
- Das Berechtigungssystem von Linux ist Benutzer-basiert, es betrifft deshalb sowohl den Speicherzugriff wie auch das Dateisystem.
- Anwendungen signieren: erschwert Code-Manipulationen und erlaubt das Teilen einer Sandbox bei gleicher sharedUser-ID
- Berechtigungen werden im Manifest deklariert, kontrollierte Öffnung der Sandbox-Restriktionen

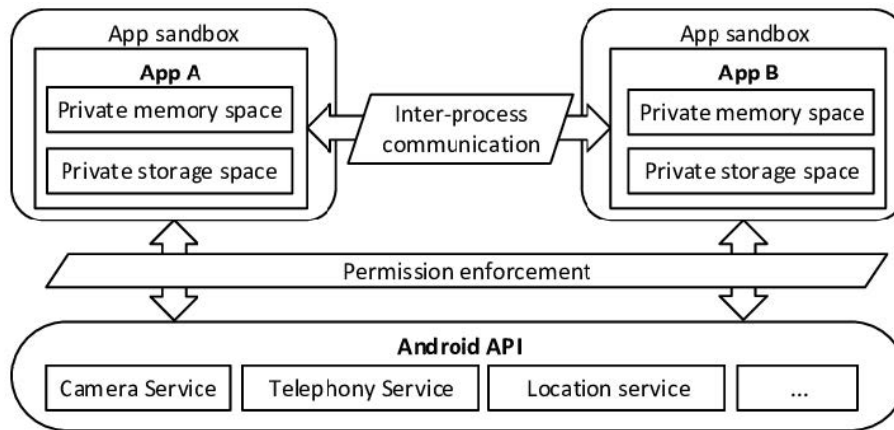


Abbildung 6: Android Security-Modell

## 2 Android 2 - Benutzerschnittstellen

### 2.1 GUI einer Activity

GUI wird als XML definiert, der Name resultiert in einer Konstante: **R.layout.xxx**. Diese wird im *onCreate()* einer Activity mit *setContentView()* angegeben.

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="@dimen/padding">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:paddingBottom="@dimen/padding"
            android:text="@string/main_title"
            android:textSize="@dimen/textSizeTitle" />

        <TextView
            android:layout_width="match_parent"
```

Abbildung 7: Beispiel eines XML für ein Layout

Je nach Layout müssen die Elemente unterschiedlich konfiguriert werden, was bei der Arbeit mit dem Layout-Editor nicht offensichtlich, aber trotzdem gut zu wissen ist.

Ein Android-UI ist hierarchisch aufgebaut und besteht aus **ViewGroups** (Container für Views oder weitere ViewGroups, angeordnet durch Layout) und **Views** (Widgets). Sollte auf unterschiedlichen Bildschirmgrößen gleich aussehen (Elemente deshalb **relativ** und nicht absolut positionieren)

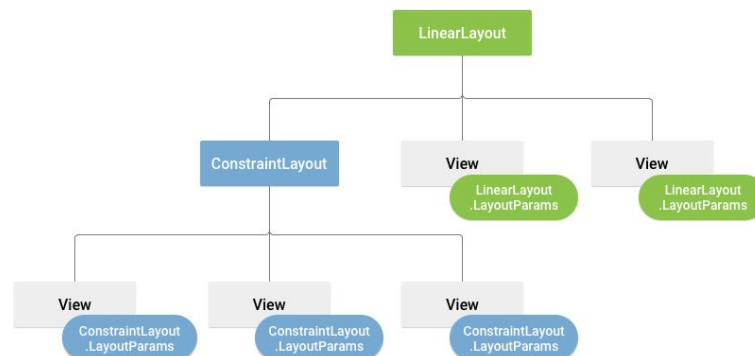


Abbildung 8: Layout-Varianten bei Android

Schachtelung möglich, aber nicht effizient, wenn möglich immer das Constraint-Layout verwenden. Layouts spezifiziert man auf zwei verschiedene Arten:

- **Statisch / Deklarativ (XML)**
- Grundsätzlich in **MOBPRO** verwendet, bietet viele Vorteile (Deklarativ, weniger umständlich als Code, Struktur eminent, Umformungen ohne Rekompilierung möglich...)
  - Deklarative Beschreibung des GUI als Komponentenbaum
  - XML-Datei unter *res/layout*
  - Referenzen auf Bilder/Texte/etc.
  - Typischerweise ein XML pro Activity
- **Dynamisch (in Java)**
- Jedes XML hat eine korrespondierende Java-Klasse, XML → Java = *Inflating*
  - Aufbau und Definition des GUI im Java-Code
  - Normalerweise nicht nötig: die meisten GUIs haben fixe Struktur
  - Änderung von Eigenschaften während Laufzeit ist normal (Bsp. Visibility, Ausblenden einer View, wenn nicht benötigt)

## 2.2 XML-Layout

- Jedes Layout ist ein eigenes XML-File
  - Root-Element = View oder ViewGroup
  - Kann Standard- oder eigene View-Klassen enthalten
- XML können mit Inflater „aufgeblasen“ bzw. instanziiert werden, damit eigene wiederverwendbare Komponenten/Templates/Prototypen erzeugt werden können
- Innere Elemente können unterhalb eines Parents via View-ID referenziert werden (*findViewById()*)
- Debugging mit dem Layout-Inspector

### 2.2.1 Constraint-Layout

- Erstellung von komplexen Layouts, ohne zu schachteln
- Elemente werden relativ mit Bedingungen platziert
  - zu anderen Elementen
  - zum Parent-Container
  - Element-Chains (spread/pack)
- Layout-Hilfen (Hilfslinien, Barriers)

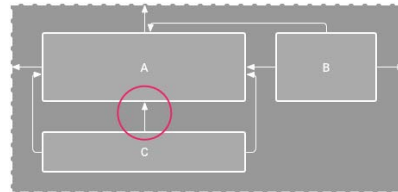


Abbildung 9: Constraint Layout

### 2.2.2 LinearLayout

- Reiht Elemente neben-/untereinander auf
  - kann geschachtelt werden, um Zeilen/-Spalten zu formen (nicht zu tief, sonst schlechte Performanz)
- Eigenschaften: (*orientation, gravity, weighthSum, etc.*)
- Layout-Parameter für Children
  - layout\_width, layout\_height
  - layout\_margin...
  - layout\_weight, layout\_gravity

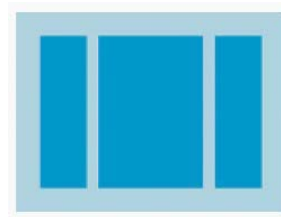


Abbildung 10: LinearLayout

### Warum nutzt man trotzdem noch LinearLayout?

- Nach wie vor einfachste Lösung für Button- oder Action-Bars („flow semantik“) und einfache Screens
- Kaum Konfiguration nötig, robust
- Für scrollbare Listen mit dynamischer Anzahl Elemente besser *ListView* verwenden (siehe Adapter-Views)
- Einsatz mit Bedacht durchaus sinnvoll

Es gibt noch die **ScrollView**, deren Nutzung vertikales Scrollen bei zu grossen Layouts erlaubt, sie kann jedoch nur ein Kind haben und enthält typischerweise das Top-Level-Layout einer Bildschirmseite.

### Pixalangaben (*Typischerweise werden Angaben in dp verwendet, ausser sp bei Schriftgrößen.*)

- **dp - density-independent:**  
Passen sich der physischen Dichte des Screens an, dp passen sich gegenüber den realen Dimensionen eines Screens und dessen Verhältnisse an.
- **sp - scale-independent:**  
Ähnlich der dp-Einheit, passt sich jedoch der Schriftskalierung des Nutzers an.
- **px - Pixels:**  
Passen sich der Anzahl Pixel eines Bildschirms an, deren Nutzung wird nicht empfohlen.

## 2.3 Ressourcen, Konfigurationen und Internationalisierung

**Ressourcen** sind alle Nicht-Java-Teile einer Applikation und sind im `/res`-Verzeichnis abgelegt, sogenannte ausgelagerte Konstanten-Definitionen. Sie werden im Layout und Java-Code über die **automatisch generierte R-Klasse** mit ID-Konstanten (int) referenziert. Kontextabhängige Ressourcen sind möglich z.Bsp. für Sprache, Gerätetyp, Orientierung, ...

**Beispiele:** Strings, Styles, Colors, Dimensionen, Bilder (drawables), Layouts (portrait, landscape), Array-Werte (z.Bsp. für Spinner) und Menü-Items

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="@dimen/marginBottom"
    android:background="@color/sectionBackground"
    android:padding="@dimen/padding"
    android:text="@string/main_section1"
    android:textColor="@color/sectionText" />
```

(a) Referenz in XML mit @

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

(b) Referenz in Code über R-Klasse, diese wird beim Build automatisch generiert

Für verschiedene Systemkonfigurationen benötigt es unterschiedliche Ausprägungen einer Ressource, beispielsweise:

- **Internationalisierung:** komplette/teilweise Übersetzung, für diese werden unterschiedliche Ordner je nach Land/Sprache und separate .xml angelegt
- **Auflösungsklassen:** ldpi ( 120dpi), mdpi ( 160dpi), hdpi ( 240dpi), xhdpi ( 320dpi)
- **Orientierung** des Displays: landscape / portrait
- Verschiedene **HW-Modelle:** HTC, Samsung, Sony, LG, ...

Default-Verzeichnisse sind innerhalb von `res/` angelegt: drawable, layout, menu, values, ...

Bei spezifischen Konfigurationen werden meist Kopien der Default-Verzeichnisse/Ordner mit einem Suffix angelegt, bspw. `res/strings-de-rCH`, in welchen dann die Ressourcen (XML) erneut angelegt werden.

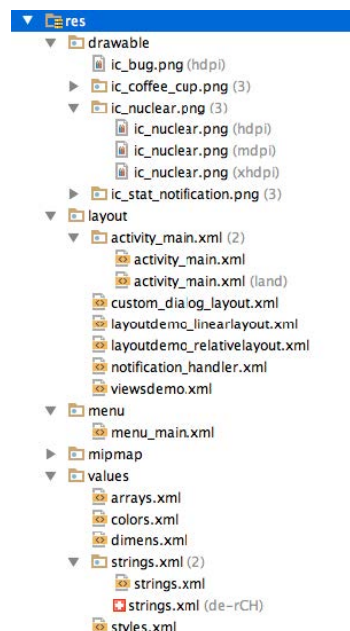


Abbildung 12: Beispiel der Default-Ressourcen

## 2.4 UI-Event-Handling

- Jedes View-Element hat eine entsprechende Java-Klasse (auch View-Groups!)  
→ Layout könnte auch dynamisch in Java programmiert werden
- APIs der einzelnen View-Klassen sind hier oder unter „Nützliche Links“ genauer beschrieben

```
<TextView
    android:id="@+id/message_label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

// Show message on dedicated text view
private void displayMessage(String message) {
    TextView label = (TextView)findViewById(R.id.message_label);
    label.setText(message);
}
```

Abbildung 13: ID im Layout erfassen und Referenz im Code

### 2.4.1 GUI-Events

- **Observer/Listener:** einen Listener für ein entsprechendes Event bei der View registrieren, bspw. bei Button myButton:  
myButton.setOnClickListener(listener)
- verschiedenste Event- und Listener-Typen:  
OnClickListener, OnLongClickListener, OnKeyListener, OnTouchListener, OnDragListener, ...  
→ public static Interfaces der Klasse View

**Ziel:** Auf Klick-Event eines Buttons reagieren

- Button muss eine ID haben im layout.xml
- Registrierung eines Listeners an die View (Button) im Code:

---

```
1 Button button = (Button) findViewById(R.id.question_button_done);
2 button.setOnClickListener(new OnClickListener() {
3     @Override
4     public void onClick(View v) {
5         // handler code
6         buttonClicked();
7     }
8 });
```

---

#### onClick-Event-Registrierung in XML

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="increaseInternalCounter"
    android:layout_marginBottom="@dimen/marginBottom"
    android:text="@string/main_increaseInternalCounter" />
```

Abbildung 14: Definition onClick-Handler im Layout → so nur für onClick-Events

---

```
1 // Implementierung onClick-Handler-Methode in der Activity
2 public void increaseInternalCounter(View button) {
3     // ... handler code ...
4 }
```

---

## 2.4.2 Exkurs: Data Binding

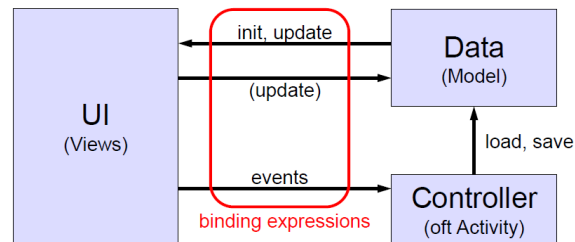


Abbildung 15: Modell für Data Binding

**Data Binding:** separiert UI und Daten, synchronisiert UI mit Daten (1-, resp. 2-way-binding), verwendet «binding expressions» mit @.. Syntax im Layout-File, um View-Attribute zu initialisieren. Anbei ein Beispiel (auskommentiert):

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <layout xmlns:android="http://schemas.android.com/apk/res/android">
3      <data>
4          <variable name="model" type="org.example.MyModel"/>
5      </data> // Definition der Layout-Variablen
6      <LinearLayout ...>
7          <Button
8              android:id="@+id/button"
9              ...
10             android:enabled="@{model.user.role == 'admin'}"
11             android:text="@{model.buttonText}" // Data Binding (1-way)
12             ...
13             android:onClick="@{() -> model.increaseClickCount()}" /> // Event Binding
14          <EditText
15              android:id="@+id/input"
16              ...
17              android:text="@={model.inputText}" /> // Data Binding (2-way)
18      </LinearLayout>
19  </layout>
20
21  protected void onCreate(Bundle savedInstanceState) {
22      super.onCreate(savedInstanceState);
23      ActivityMainBinding binding = DataBindingUtil.setContentView(...);
24      model = new MainModel();
25      model.load();
26      binding.setModel(model);
27      // Binden der Layout-Daten auf effektive Daten
28      // z.B. ViewModel mit Observables
29  }
```



## 2.5 Options-Menü

- Android-Apps können oben rechts ein Menü mit Optionen anbieten
- Erzeugung durch Aufruf *Hook* in der Activity-Klasse:

`onCreateOptionsMenu(Menu menu)`

- Hier kann ein Menü mit Einträgen bestückt werden
- `MenuInflater` + XML benutzen oder Java oder beides

- Beim Klick auf Eintrag Aufruf eines anderen Hooks:

`onOptionsItemSelected(MenuItem item)`

Für ein Options-Menü muss eine .xml-Datei (Bsp. `main_menu.xml`) im Ordner `res/menu` angelegt werden. Danach werden Informationen folgendermassen eingetragen:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity">
  <item
    android:id="@+id/main_menu_finish"
    android:title="@string/menu_finish">
  </item>
  <item
    android:id="@+id/main_menu_startAllViews"
    android:title="@string/menu_startViewsDemo">
  </item>
</menu>
```

(a) Menü und Items in XML definieren

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_main, menu);
    return true;
}
```

(b) Menü mit `MenuInflater` aufblasen

Um bspw. einen String in einem Menüpunkt einzufügen, gibt es drei verschiedene Möglichkeiten:

```
menu.add(Menu.NONE, 239, Menu.NONE, "Menu Item 1");
menu.add(Menu.NONE, 333, Menu.NONE, getString(R.string.menu_mail));
menu.add(Menu.NONE, 923, Menu.NONE, R.string.menu_server);
```

Abbildung 17: Möglichkeiten zum Einlesen eines Strings

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (super.onOptionsItemSelected(item)) {
        return true; // handled by super implementation
    }
    switch (item.getItemId()) {
        case R.id.main_menu_finish:
```

Abbildung 18: Event-Handling: Selektierung

## 2.6 Adapter-Views

Behandelt wird hier nur das synchrone Laden von kleinen/schnellen Datenquellen, für asynchrones Laden von langsamen/grossen Datenquellen konsultiere Doku über **Loaders**.



Abbildung 19: Aufgabe des Adapters

- Adapter → Verbindung zwischen Datenquelle und GUI
- Zapft *Datenquelle* an und beliefert *AdapterView*
- Erzeugt (Sub-)Views pro gefundenes Datenelement
- Transformiert Daten ggf. in benötigtes Zielformat
- Datenquellen:  
String-Array, String-Liste, Bilder, Datenbank, ...

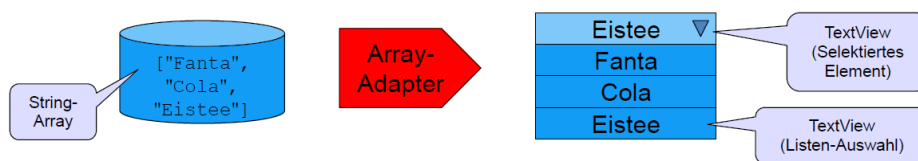


Abbildung 20: Beispiel eines ArrayAdapter

- Bindet irgend ein Array oder Liste mit beliebig getypeten Elementen an irgend eine AdapterView
- Für jedes Daten-Element wird eine SubView erzeugt
- **Default:** Erstellt `TextView` mit `element.toString()`-Wert

```
String[] myArray = new String[]{"Fanta", "Cola", "Eistee"};
ArrayAdapter<String> adapter =
    new ArrayAdapter<String>(this, android.R.layout.simple_list_item_checked, myArray);
this.setListAdapter(adapter);
```

Abbildung 21: Beispiel einer AdapterView

### 2.6.1 AdapterViews & ListActivity

- **AdapterViews:** spezielle View-Klassen
  - Sind für Zusammenarbeit mit Adaptern optimiert  
(Bsp. `ListView`, `GridView`, `Gallery`, `Spinner`, `Stack`, ...)
  - Füllen Teile von sich mit von Adaptern erzeugten Views
  - Leiten ab von `android.widget.AdapterView<T> extends android.widget.Adapter<>`
- Spezielle Activity: **ListActivity**
  - Vordefiniertes Layout (enthält eine `ListView`, kein XML nötig)
  - Vordefinierte Callbacks (bei Auswahl einer List-Entry)
  - Bietet Zugriff auf aktuelle Selektion / Datenposition

## 2.6.2 android.widget.Spinner

- ComboBox oder DropDown-List genannt (weitere Alternative: AutoCompleteTextView)
- Zeigt ein ausgewähltes Element, bei Klick erscheint ein Auswahlmenü
- 2 Varianten, um Daten auf Spinner zu setzen:
  - Im Code mit Adapter:  
`spinner.setAdapter(myAdapter)`
  - Im XML mit Angabe einer String-Array-ID:  
`android:entries="@array/spinnerValues"`
- Listener setzen für Behandlung der Auswahl:  
`spinner.setOnItemClickListener(...)`

## Demo: Spinner (Siehe Übung 2)

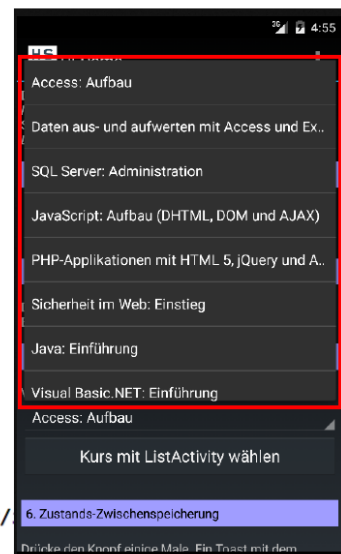
### ■ layout.xml

```
<Spinner
    android:id="@+id/main_spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:entries="@array/itCourses"
    android:prompt="@string/main_spinnerPrompt" />
```

Beachte: Daten werden aus XML-Ressource geholt

### ■ arrays.xml

```
<resources>
    <string-array name="itCourses">
        <item>Access: Aufbau</item>
        <item>Daten aus- und aufwerten mit Access und Excel</item>
        <item>SQL Server: Administration</item>
        <item>JavaScript: Aufbau (DHTML, DOM und AJAX)</item>
        <item>PHP-Applikationen mit HTML 5, jQuery und AJAX</item>
        <item>Sicherheit im Web: Einstieg</item>
        <item>Java: Einführung</item>
    </string-array>
</resources>
```



### ■ In der Activity-Klasse

```
spinner.setOnItemClickListener(new OnItemSelectedListener() {
    public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
        String selectedItem = (String) parent.getItemAtPosition(position);
    }
});
```

Position der View in „ParentView“

Zeilen-ID des gewählten Werts bei DB-Query

Abbildung 22: Übungs-Demo aus der Vorlesung SW02 - Spinner

### 2.6.3 android.widget.ListView

- Liste von Views/Items, die zur Auswahl stehen
- Braucht viel Platz! Meist wird ihr der ganze Bildschirm zugeteilt
- i.d.R. zusammen mit `ListActivity` verwendet, Verwendung:
  1. Navigiere zu eigener `ListActivity`
  2. Auswahl → Resultat setzen → finish
  3. Auswertung des Rückgabewert im Caller
- Konzeptionell identisch zum Spinner, jedoch andere Darstellung auf UI
  - Verwendungsentscheid:
    - \* Kurze Liste → Spinner
    - \* (Sehr) lange Listen → `ListView` / `ListActivity`
    - \* Kennt der User die möglichen Auswahlwerte → `AutoCompleteTextView`
  - Adapter- / Datendefinition grundsätzlich bei beiden gleich (d.h. im Code oder durch XML-Array)
  - Auswahlmodus: `setChoiceMode(ListView.CHOICE_MODE_*)`  
→ Single- / Multiselection

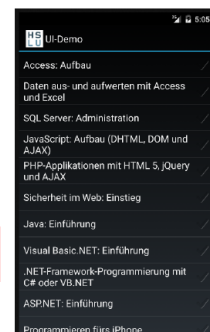
### 2.6.4 android.app.ListActivity

- Spezielle Activity zur Darstellung einer `ListView`
- Vordefiniertes Layout (full-screen Liste)
  - `setContentView(...)` muss nicht aufgerufen werden
  - Aufruf i.d.R. mit `startActivityForResult(...)`
  - Vordefinierte vererbte Konfigurationsmethoden
    - \* `setListAdapter(adapter)` setzt Daten für die Liste
    - \* `getListView()` erlaubt Zugriff auf `ListView`-Instanz (anstelle von `findViewById(..)` + Casten)
- Callback bei der Auswahl
  - `onListItemClick(parentView, view, position, id)`  
Wird bei Auswahl aufgerufen (muss in Subklasse überschrieben werden, keine Listener-Registrierung nötig)

## Demo: ListView & ListActivity (Siehe Übung 2)

- Activity-Klasse (erbt von `ListActivity`!)
- Initialisierung der `ListActivity` mit Daten

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Attention: We do NOT set a layout! - ListActivity has already defined a layout.
    String[] courses = getResources().getStringArray(R.array.itCourses);
    ArrayAdapter<String> adapter =
        new ArrayAdapter<String>(this, android.R.layout.simple_list_item_checked, courses);
    this.setAdapter(adapter);
    ListView listView = getListView();
    listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
}
```



- Reagieren auf Selektion

```
@Override
protected void onListItemClick(ListView parent, View view, int position, long id) {
    // define return value
    Intent result = new Intent();
    String selectedItem = (String) parent.getItemAtPosition(position);
    result.putExtra(EXTRA_CLASS_KEY, selectedItem);
    // set return value
    setResult(RESULT_OK, result);
    // finish the activity
    finish();
}
```

Position der View in „ParentView“

Zeilen-ID des gewählten Werts bei DQ-Query

Abbildung 23: Übungs-Demo aus der Vorlesung SW02 - `ListView` / `ListActivity`

## 2.7 ViewModel - Konfigurationswechsel & temporäre Datenspeicherung

Bei jedem Konfigurationswechsel (z.B. Änderung Bildschirmorientierung) wird die aktuelle Activity-Instanz zerstört und neu aufgebaut. Dabei besteht das Problem des **Zustandsverlusts**. Der Zustand aller Views mit einer ID (mit einigen Ausnahmen) wird automatisch gesichert und wiederhergestellt. Der **inhärente Zustand**, alles was nicht sichtbar und in Feldern gespeichert ist, geht jedoch verloren. Um entgegenzuwirken, kann ein **ViewModel** verwendet werden.

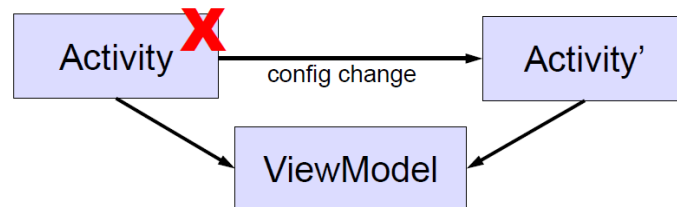


Abbildung 24: Position des ViewModels in der temp. Datenspeicherung

- Kapselt UI-Daten so, dass diese bei einer Konfigurationsänderung einer Activity in-memory erhalten bleiben (Für den Fall eines App-Kills müssen Daten immer noch persistiert werden)
- Lebensdauer mit der Activity gekoppelt
- Weniger Aufwand für Behandlung von Konfigurationsänderungen

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0' // ViewModel and LiveData
    ...
}
```

Zusätzliche Gradle dependency für ViewModel und Lifecycle Management

```
public class MainViewModel extends ViewModel {
    private int counter = 0;

    public int incrementCounter() { return ++counter; }

    public int getCounter() { return counter; }
}
```

ViewModel = normales POJO, ggf. mit Handler-Methoden

Wäre noch viel einfacher mit DataBinding! (out-of-scope)

```
// in MainActivity
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    viewModel = ViewModelProviders.of(this).get(MainViewModel.class);

    counterLabel = findViewById(R.id.main_label_counter);
    updateCounterLabel();

    // called on button click (see main.xml)
    public void increaseInternalCounter(View button) {
        viewModel.incrementCounter();
        updateCounterLabel();
    }
}
```

Erzeuge oder hole ViewModel-Instanz für diese Activity-Lebenszyklus-Instanz

Initialisierung UI aus ViewModel

**Demo**

Abbildung 25: Übungs-Demo aus der Vorlesung SW02 - ViewModel

## 2.8 Rückmeldungen an den Benutzer

### 2.8.1 Toast

- Kurze Rückmeldung (Popup) an den Benutzer, keine Interaktion möglich, verschwindet nach gewisser Zeit.
- Konfiguration: Text, Layout, Anzeigzeit (kurz/lang), Ort (gravity)
- Toasts mit eigenem Layout werden mit CustomToastView erstellt

Beispielcode zur Erstellung von Toasts:

```
1 // Default-Toast: Einzeiler
2 Toast.makeText(getApplicationContext(), "Das ist..", Toast.LENGTH_LONG).show()
3                                     // LENGTH: Nur LONG oder SHORT
4                                     // Kontext: meistens "this"
5
6 // Toast mit anderem Anzeigort:
7 Context context = getApplicationContext();
8 Toast toast = Toast.makeText(context, "Toast links oben!", Toast.LENGTH_LONG);
9 toast.setGravity(Gravity.TOP|Gravity.START, 0, 0); // (x,y) Offset
10 toast.show()
```

### 2.8.2 Alert-Dialog

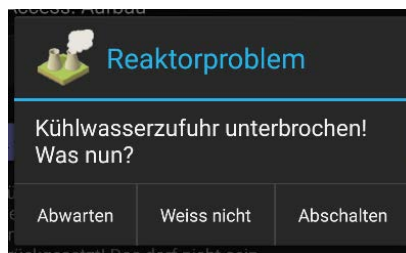


Abbildung 26: Beispiel eines Alert-Dialogs

- Fenster mit Interaktionsmöglichkeiten für den Benutzer
  - Information / Eingabe von Daten
  - Interaktion möglich
  - Buttons: positive, neutral, negative
- Vorteile
  - Kaum Einschränkungen in puncto Darstellung
  - Vorbereitet für die Anzeige von Daten
  - Verschwindet erst, wenn sie vom Benutzer quittiert wurde
- Konfiguration: Buttons, Titel, Icon, Nachricht  
Inhalt: Liste von Items oder eigene View

- Vorgehen beim Erstellen eines Alert-Dialog mit Builder-Muster
  1. Builder erstellen: `new AlertDialog.Builder(this)`
  2. Builder konfigurieren:  
`setXXX + Registrierung von ClickListeners`
  3. Dialog erstellen: `Dialog dialog = builder.create()`
  4. Dialog anzeigen: `dialog.show()`
- Anzeige von Dialogen ist **immer asynchron!**  
Bei `show()` wird nicht gewartet, kein Rückgabewert  
→ Behandlung von Benutzerselektion mit Listener

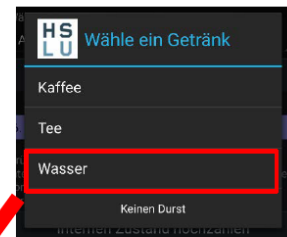
```
AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
dialogBuilder.setTitle("Reaktorproblem")
    .setIcon(R.drawable.ic_nuclear)
    .setMessage("Kühlwasserzufuhr unterbrochen!\nWas nun?")
    .setPositiveButton("Abschalten", new OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            Toast.makeText(getApplicationContext(),
                "Reaktor wird abgeschaltet...",
                Toast.LENGTH_LONG).show();
        }
    }).setNeutralButton("Weiss nicht", new OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            Toast.makeText(getApplicationContext(),
                "Problem an Support weitergeleitet...",
                Toast.LENGTH_SHORT).show();
        }
    }).setNegativeButton("Abwarten", new OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            // do nothing
        }
    });
return dialogBuilder.create();
```

Abbildung 27: Beispiel eines AlertDialog aus Vorlesung



## Alert-Dialog mit Auswahl-Daten

- Titel, Icon, usw. wie gehabt
- Neu: Daten (Array) setzten
  - Methode `setItems(...)`
    - Inkl. ClickListener
      - Toast mit Wahl anzeigen!



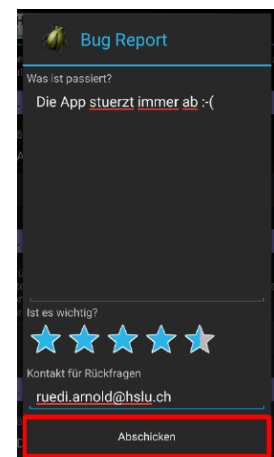
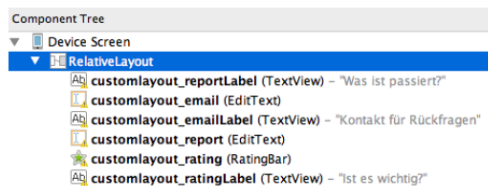
```
final String[] items = {"Kaffee", "Tee", "Wasser"};
AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
dialogBuilder.setTitle("Wähle ein Getränk")
    .setIcon(R.mipmap.ic_launcher)
    .setItems(items, new OnClickListener() {
        public void onClick(DialogInterface dialog, int itemPos) {
```

Handler für  
Selection

Abbildung 28: Beispiel mit Auswahl-Daten

## Alert-Dialog mit eigenem Layout

- Layout.xml „aufblasen“ & setzen



```
private Dialog createCustomLayoutDialog() {
    final View customView = LayoutInflater.from(this).inflate(
        R.layout.custom_dialog_layout, null);
    AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
    dialogBuilder.setTitle("Bug Report").setIcon(R.drawable.ic_bug)
        .setView(customView)
        .setPositiveButton("Abschicken", new OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
```

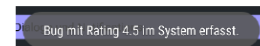


Abbildung 29: Beispiel mit eigenem Layout



Ein (offener) Dialog gehört zum Zustand einer Activity, ist ein Dialog noch geöffnet bei einem Konfigurationswechsel, dann wird dieser nicht gespeichert und auch nicht wiederhergestellt! Deshalb sollten Dialoge als `DialogFragment` implementiert werden. Der Zustand des Dialogs wird dann vom `FragmentManager` korrekt mit Lifecycle und Activity synchronisiert (save/restore)

Für den Moment: Ein **Fragment** ist ein wiederverwendbarer „UI Schnippsel“ mit eigenem Zustand und Lifecycle.

### 2.8.3 Notifications (Status-Bar)

- Persistente Nachricht
  - Kurze Ticker-Nachricht in der Status-Bar
  - Danach persistente Anzeige im Notification Window
  - Bei Auswahl erfolgt Aufruf einer definierten Activity
- Vorteile:
  - Nachricht bleibt erhalten bis vom Nutzer quittiert
  - Beliebig komplexe Behandlung, da Start einer Activity
- Nachteil:
  - Etwas komplexere Mechanik wegen `PendingIntent`

## Demo: Notification

### ■ Code verwendet u.a.:

- `AlertDialog.Builder`
- `Notification.Builder`
- `PendingIntent`
- `NotificationManager`
- Eigene dedizierte Activity für Darstellung der Nachricht

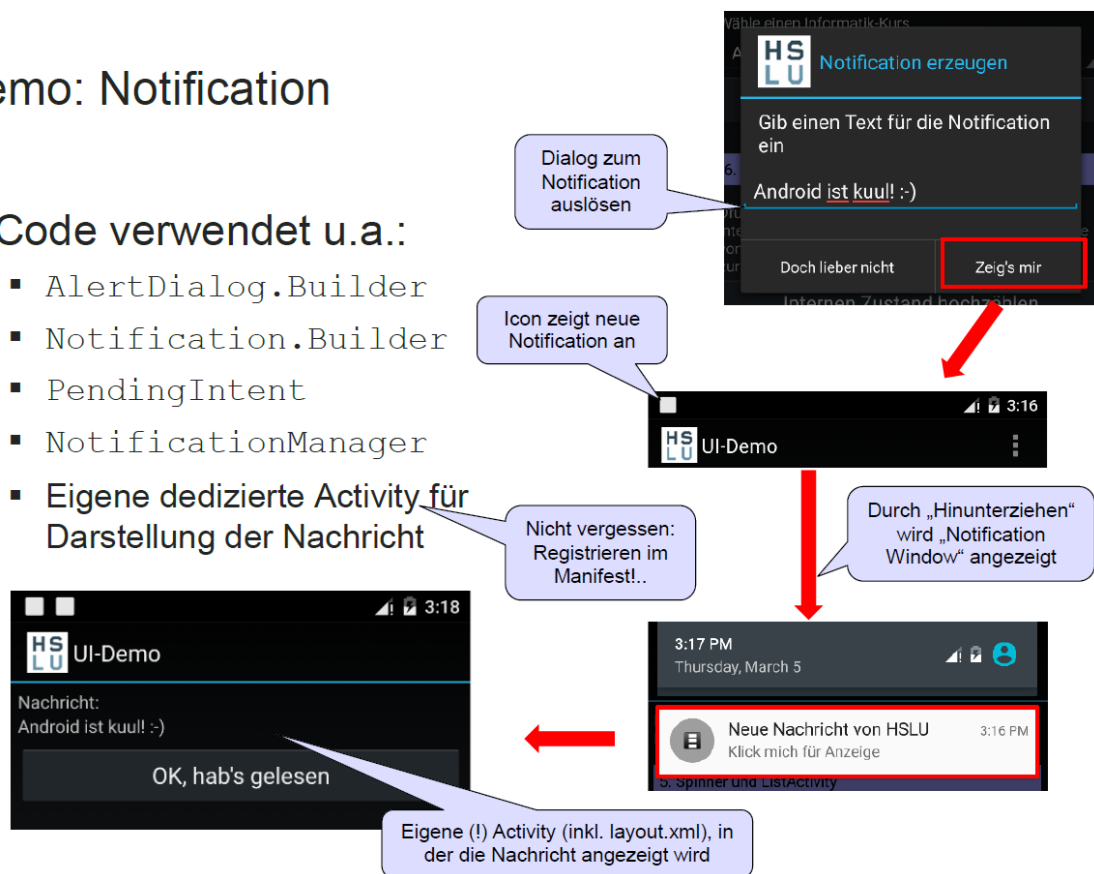


Abbildung 30: Übungs-Demo aus der Vorlesung SW02 - Notification

## 3 Android 3 - Persistenz & Content Providers

Persistenz: Daten über Laufzeit der App erhalten. Für lokale Persistenz gibt es drei Möglichkeiten:

- **Shared Preferences**  
Key/Value-Paare, Verwendung für kleine Datenmengen
- **Dateisystem**  
intern oder extern, in App-Sandbox (privat) oder auf SD-Karte (öffentlich), Verwendung für binäre/-grosse Dateien, Export
- **Datenbank (Room)**  
SQLite + Object Relational Mapper (ORM), Verwendung für strukturierte Daten + Abfragen/Suche

### 3.1 (Shared) Preferences

- Jede Activity hat ein SharedPreferences-Profil, persistente Einstellungen für Activity oder Applikation
- Key-Value-Store (persistente Map)
- Preferences für **Activity**:  
`Activity.getSharedPreferences(mode)`  
Anwendungsfall: Activity-State persistent speichern
- Preferences für **Applikation**:  
`PreferenceManager.getDefaultSharedPreferences(ctx)`  
`Context.getSharedPreferences(name, mode)`
- Mögliche Datentypen für Preferences-Werte:  
*String, int, float, long, boolean, Set<String>* (mit separaten Werten)

#### Lesen und Schreiben auf Preferences

- Mehrere Dateien pro Applikation möglich, Zugriff mit  
`Activity.getSharedPreferences(name, mode)` (unterschiedliche Dateinamen)  
oder auch über `getDefaultSharedPreferences(mode)`, die Applikation findet danach anhand der Preference-Benennungen die Einträge auch selber
- Lesen mit Methoden `SharedPreferences.getX()`  
**X** steht für den Typ, also String, Int, Boolean, ...
- Schreiben immer mit dem Editor:
  1. `SharedPreferences.Editor editor = preferences.edit()`
  2. `editor.putX(...)`
  3. `editor.apply()` Persistierung der Änderungen
    - asynchrone Persistierung, blockiert die Methode nicht
    - für synchrone Persistierung: `editor.commit()`

Beispiel, um die Anzahl Aufrufe einer App über die Lebenszeit der App hinaus zu persistieren:

---

```
1 final SharedPreferences preferences = getPreferences(MODE_PRIVATE);
2 final int newResumeCount = preferences.getInt(COUNTER_KEY, 0) + 1;
3 final SharedPreferences.Editor editor = preferences.edit();
4 editor.putInt(COUNTER_KEY, newResumeCount);
5 editor.apply();
```

---

#### 3.1.1 Darstellung User-Preferences

- Automatische Darstellung mit `PreferenceFragment`, eigener Editor für jeden Wertetyp
- `PreferenceFragment` schreibt/liest grundsätzlich in die `DefaultSharedPreferences`, kann aber auch für andere Preference-Stores konfiguriert werden

User-Präferenzen können in XML deklariert werden unter `res/xml` z.Bsp. als `preferences.xml`, wobei untersch. Präferenzen bspw. als `CheckBoxPreference`, `ListPreference` usw. erfasst werden. Daten können wie in diesem Beispiel aus den Array-Ressourcen bezogen werden:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:key="teaPrefs"
        android:title="Tee Präferenzen">

        <CheckBoxPreference
            android:key="teaWithSugar"
            android:persistent="true"
            android:summary="Soll der Tee gesüsst werden?"
            android:title="Gesüsster Tee?" />

        <ListPreference
            android:dependency="teaWithSugar"
            android:entries="@array/teaSweetener"
            android:entryValues="@array/teaSweetenerValues"
            android:key="teaSweetener"
            android:persistent="true"
            android:shouldDisableView="true"
            android:summary="Womit soll der Tee gesüsst werden?"
            android:title="Süsstoff" />

        <EditTextPreference
            android:key="teaPreferred"
            android:persistent="true"
            android:summary="z.B. "Lipton/Pfefferminztee""
            android:title="Bevorzugte Marke/Sorte" />

    </PreferenceCategory>
</PreferenceScreen>
```

Abbildung 31: Beispiel eines Präferenzen-XML

- Ohne `android:summary` würde die gewählte Preference angezeigt werden
- `android:dependency` deklariert eine Abhängigkeit zu einer anderen Preference, ist diese nicht gegeben kann die andere Preference nicht ausgewählt werden
- **Entries:** „Anzeigestring“, übersetzbar  
**EntryValues:** „Werte“, nicht übersetzt, technischer Schlüssel

---

```
1 // Zur "Uebersetzung" von Values zu Entries (Beispiel)
2 public String getValueFromKey(String key) {
3     String[] keys = getResources().getStringArray(R.array.teaSweetenerValues);
4     String[] values = getResources().getStringArray(R.array.teaSweetener);
5     int i = 0;
6     while(i < keys.length) {
7         if(keys[i].equals(key)) {
8             return values[i];
9         }
10        i++;
11    }
12    return "";
13 }
```

---

### 3.1.2 PreferenceFragment

Ein PreferenceFragment kann in einer eigenen Activity (hier TeaPreferenceActivity) erstellt werden:

---

```
1 public class TeaPreferenceActivity extends Activity {
2
3     @Override
4     protected void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         getFragmentManager().beginTransaction().replace(android.R.id.content,
7             new TeaPreferenceInitializer()).commit();
8     }
9
10    // PreferenceFragment als statische innere Klasse
11    public static final class TeaPreferenceInitializer extends PreferenceFragment
12    {
13        @Override
14        public void onCreate(final Bundle savedInstanceState) {
15            super.onCreate(savedInstanceState);
16            addPreferencesFromResource(R.xml.preferences);
17            // Referenz auf preferences.xml
18        }
19    }
```

---

### 3.1.3 Default-Präferenzen

Präferenzen können programmatisch auch wieder auf „Standard“-Werte oder auf festgelegte Werte gesetzt werden, für das Tee-Beispiel kann dies bspw. folgendermassen vorgenommen werden:

---

```
1 SharedPreferences teaPrefs = PreferenceManager.getDefaultSharedPreferences(this);
2 SharedPreferences.Editor editor = teaPrefs.edit();
3 editor.putString("teaPreferred", "Lipton/Pfefferminztee");
4 editor.putString("teaSweetener", "natural");
5 editor.putBoolean("teaWithSugar", true);
6 editor.apply();
```

---

## 3.2 Dateisystem

- **Einsatzbereiche**
  - Speichern/Laden von binären Dateien (Bilder, Musik, Video, Java-Objects, etc.)
  - Caching (Heruntergeladene Dateien)
  - Grosse Text-Dateien (Plain Text, Strukturierte Daten wie XML, JSON, etc.)
- Teilen / Freigeben von erstelltem Inhalt (Externer Speicher wie SD-Karte)
- Dateien sind entweder
  - PRIVATE → ins Applikationsverzeichnis  
(Zugriff für andere Apps nur über Content Provider möglich)
    - \* `Context.getFilesDir()`
  - PUBLIC → auf die SD-Karte
    - \* `Environment.getExternalStorageDirectory()`  
`Environment.getExternalStorageState();`
- Für Zugriff auf SD-Karte muss die Permission im Manifest eingetragen werden! (siehe nachfolgend)

### 3.2.1 Exkurs: Permission-Model

- Vor gewissen Operationen müssen Apps die Berechtigung des Nutzers erhalten (Kontaktzugriff, Internet, SD-Karte, Kamera, SMS, etc.)
- Klasse: `android.Manifest.permission`
- Seit API 23 werden keine dangerous Permissions mehr gewährt, der Nutzer muss diese selber freigeben (Applikation fragt beim Nutzer nach), Permissions werden einzeln gewährt/abgelehnt.  
**Konsequenz:** Apps müssen mit eingeschränkten Permissions umgehen können
- Arten von Permissions
  - *normal*
    - \* Wird bei der Installation automatisch erlaubt
  - *dangerous*
    - \* Muss von User erlaubt werden (kann wieder entzogen werden)
  - *signature*
    - \* Wird automatisch erlaubt, wenn die App, welche die Permission definiert, vom gleichen Hersteller ist wie die App, welche die Permission beanträgt (sonst ist sie „dangerous“)
  - *signatureOrSystem*
    - \* Wird automatisch erlaubt für Apps, welche im System-Image sind, sonst wie „signature“
- Permissions können gruppiert werden, User gibt Freigabe für alle Permissions in einer Gruppe (keine einzelnen Permissions), falls benötigt

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    package="ch.hslu.mobpro.persistence"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.WRITE_SMS" />

</manifest>
```

Abbildung 32: Erfassung von Permissions im Manifest

### 3.2.2 Exkurs ff: Runtime Permissions

```
public void loadExtFileWithPermission() {  
    int grant = checkSelfPermission(Manifest.permission.READ_EXTERNAL_STORAGE);  
    if (grant != PackageManager.PERMISSION_GRANTED) {  
        requestPermissions(new String[]{ Manifest.permission.READ_EXTERNAL_STORAGE }, 24);  
    } else {  
        // permission already granted  
        readFile();  
    }  
}
```

Abbildung 33: RuntimeCheck der Permissions

```
@Override  
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {  
    switch (requestCode) {  
        case 24: // load file  
            if (grantResults.length > 0 && grantResults[0] != PackageManager.PERMISSION_GRANTED) {  
                Toast.makeText(this, "Permission " + permissions[0] + " denied!", Toast.LENGTH_SHORT).show();  
            } else {  
                // permission was granted  
                loadFile();  
            }  
            break;  
    }  
}
```

Abbildung 34: Callback aus Permission-Abfrage

### 3.2.3 Exkurs ff: Persistenz mit Datei

Repetition zu Streams, Reader & Co.

- Stream: Byte-Datenstrom [28, 11, 200, 255, 2, 15, 33]
  - Auf File öffnen:  
FileOutputStream, FileInputStream
- Stream kann in Zeichenstrom ['h', 'a', 'l', 'l', 'o'] umgewandelt werden
  - FileReader, FileWriter + „Buffered“-Versionen
- Immer schliessen!  
stream.close() / reader.close()
- Nicht vergessen: try-catch-finally implementieren
- java.nio.file.Path: ist ab API 26 in Android verfügbar!

```
Writer writer = null;  
try {  
    writer = new BufferedWriter(new FileWriter(outFile));  
    writer.write(text);  
    return true;  
} catch (final IOException ex) {  
    // ...  
} finally {  
    Log.e("HSLU-MobPro-Persistenz", "Got a problem");  
    // ...  
}
```

Abbildung 35: Beispielcode zur Persistierung in einem Textfile

### 3.3 Datenbank (Room)

Android-DB **SQLite** ist bei Android fix integriert. Ein DB-Adapter ist die Verbindung zwischen Business-Objekten und einer Datenbank.

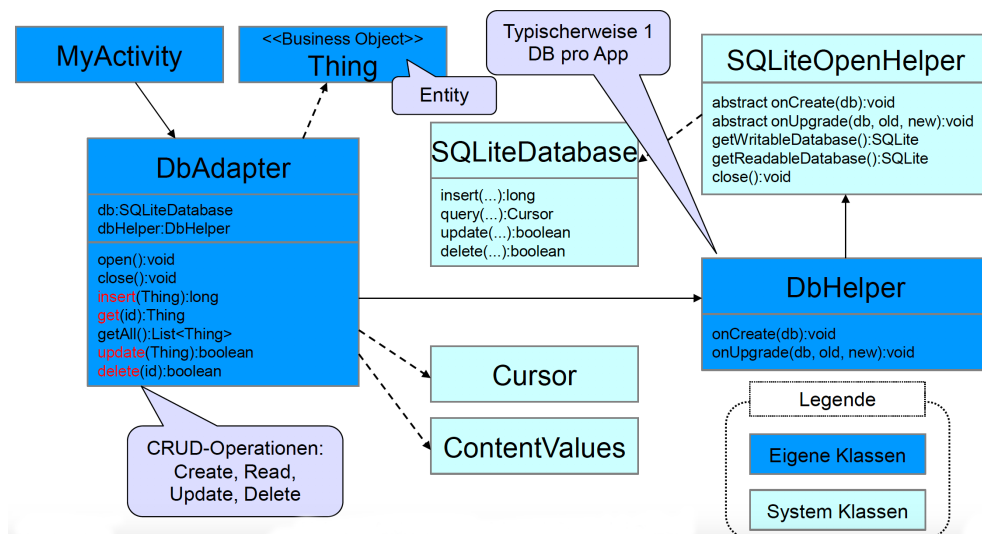


Abbildung 36: SQLite Framework

- Room ist ein Object Relational Mapper (ORM) für Android
  - Klassen werden auf relationale DB-Tabellen gemappt
  - Zugriff auf Datenbank wird abstrahiert
    - Typischerweise werden SQL-Statements durch Methodenaufrufe gekapselt
- Spezialfälle des Room ORM
  - Datenzugriff über DAO*: Queries werden als SQL-Statements in Annotationen definiert
  - Beziehungen zwischen Entitäten müssen manuell abgebildet werden (Performance!)
  - Nested Objects*: mehrere POJOs in einer Tabelle
  - Einschränkungen für Datenzugriffe, standardmässig nicht möglich im UI Thread
- Die drei Komponenten von Room
  - Database** Abstraktion der Datenverbindung
  - Entity** Repräsentation einer Tabelle in der relationalen DB
  - DAO** (Data Access Object) Enthält Methoden für Datenzugriff

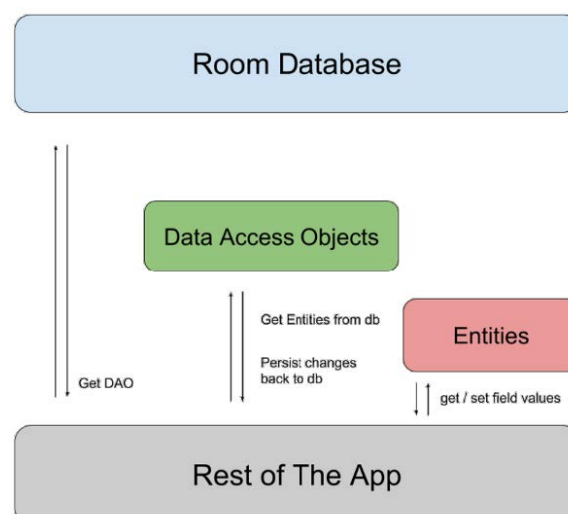


Abbildung 37: Komponenten von Room

### 3.3.1 Room - Code-Beispiele

---

```
1 @Entity // POJO mit Annotationen
2 public class User {
3     @PrimaryKey
4     public int uid;
5
6     @ColumnInfo(name = "first_name")
7     public String firstName;
8
9     @ColumnInfo(name = "last_name")
10    public String lastName;
11 }

```

---

```
1 @Dao // Datenzugriff ueber Annotationen (teilweise mit SQL-Queries)
2 public interface UserDao {
3     @Query("SELECT * FROM user")
4     List<User> getAll();
5
6     @Query("SELECT * FROM user WHERE uid IN (:userIds)")
7     List<User> loadAllByIds(int[] userIds);
8
9     @Insert
10    void insertAll(User... users);
11
12    @Delete
13    void delete(User user);
14 }

```

---

```
1 // Database: Subklasse von RoomDatabase, konfiguriert mit Database Annotation
2 @Database(entities = {User.class}, version = 1)
3 public abstract class AppDatabase extends RoomDatabase {
4     public abstract UserDao userDao();
5 }
6
7 // Zum Erzeugen einer Instanz der DB:
8 AppDatabase db = Room.databaseBuilder(
9     getApplicationContext(),
10    AppDatabase.class,
11    "database-name"
12 ).build();

```

---



### 3.3.2 Room - Daten mit Entitäten definieren

- Ein POJO mit @Entity Annotation
- Primärschlüssel (wird in jeder Entität benötigt)
  - @PrimaryKey für ein einzelnes Feld, optional mit autoGenerate Property
  - Für zusammengesetzte Primärschlüssel: primaryKeys Property in @Entity Annotation
- Falls bestimmte Felder nicht gespeichert werden sollen
  - @Ignore Annotation für ein einzelnes Feld
  - mit ignoredColumns Property in @Entity Annotation für mehrere Felder (v.a. von Superklassen)

---

```
1 // Code-Beispiel
2 // ACHTUNG: dieses Beispiel definiert mehrere Primary Keys und vermischte
  // Ansätze zum Ignorieren von Feldern zwecks Syntax-Demonstration!
3
4 @Entity(primaryKeys = {"firstName", "lastName"},
5         ignoredColumns = {"password", "otherField"})
6 public class User extends Party {
7     @PrimaryKey(autoGenerate = true)
8     public int id;
9
10    public String firstName;
11    public String lastName;
12
13    @Ignore
14    Bitmap picture;
15 }
```

---

### 3.3.3 Room - Beziehungen modellieren

**Beachte:** Room erlaubt aus Performanzgründen keine Objektreferenzierungen!

---

```
1 @Entity(foreignKeys = @ForeignKey(entity = User.class,
2                                   parentColumns = "id",
3                                   childColumns = "user_id"))
4 public class Book {
5     @PrimaryKey
6     public int bookId;
7
8     public String title;
9
10    @ColumnInfo(name = "user_id")
11    public int userId; // Feld-Typ: nur ID, nicht User
12 }
```

---

### 3.4 Mit DAOs auf Daten zugreifen

- DAOs enthalten Methoden für den abstrahierenden Datenbankzugriff
- Trägt zur *Separation of Concerns* bei und erhöht die Testbarkeit  
→ DAOs können gemockt werden!
- DAOs als Interfaces oder abstrakte Klassen definieren  
→ Room erzeugt passende Implementationen bei der Kompilierung  
(Typischerweise eine DAO-Klasse pro Entity, mit allen möglichen Operationen)
- Zwei Möglichkeiten:  
Convenience Queries **oder** `@Query` Annotation mit SQL-Statements

#### 3.4.1 Convenience Queries

- Werden über Annotations für die jeweiligen Methoden definiert:  
`@Insert`, `@Update`, `@Delete`
- Alle Parameter müssen Klassen mit einer `@Entity` Annotation  
(oder Collections/Arrays) davon sein
- Rückgabewerte:
  - **Insert:** `long` bzw. `long[]` bzw. `List<Long>` (liefert Row-IDs zurück)
  - **Update / Delete:** `int` (Anzahl modifizierte Tabelleneinträge)

---

```
1 @Insert
2 public long[] insertUsersAndFriends(User user, List<User> friends);
3     // ID Rueckgabe, sonst void
4                                     // Parameter fuer Operation (Entities)
```

---

```
1 // Weitere Convenience Queries Beispiele
2
3 @Dao
4 public interface MyDao {
5
6     @Insert (onConflict = OnConflictStrategy.REPLACE)
7     public void insertUsers(User... users);
8
9     @Insert
10    public void insertBothUsers(User user1, User user2);
11
12    @Insert
13    public long[] insertUsersAndFriends(User user, List<User> friends);
14
15
16    @Update
17    public void updateUsers(User... users);
18
19    @Delete
20    public void deleteUsers(User... users);
21 }
```

---

### 3.4.2 Custom Queries mit @Query

- Die @Query Annotation kann für Schreib- und Lesevorgänge genutzt werden
- Jede @Query wird zur Kompilierzeit überprüft  
→ Kompilierfehler bei ungültigen Queries, keine Laufzeitfehler!
- Für eine @Query kann eine beliebige Anzahl (0..n) Parameter verwendet werden
- Wenn nicht ganze Objekte benötigt werden, können Ressourcen gespart werden durch die Verwendung von POJOs mit @ColumnInfo Annotationen

---

```
1 // Custom Queries Codebeispiele
2
3 @Dao
4 public interface MyDao {
5     @Query("SELECT * FROM user")
6     public User[] loadAllUsers();
7
8     @Query("SELECT * FROM user WHERE age > :minAge")
9     public User[] loadAllUsersOlderThan(int minAge);
10
11     @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
12     public List<NameTuple> loadUsersFromRegions(List<String> regions);
13 }
14
15 public class NameTuple {
16     @ColumnInfo(name = "first_name")
17     public String firstName;
18
19     @ColumnInfo(name = "last_name")
20     public String lastName;
21 }
```

---

### 3.5 DB-Einträge in einer Liste anzeigen

- Verschiedene Ansätze, je nach Umfang/Komplexität der Datensätze: ListView, RecyclerView, Kombination mit ViewModel und LiveData
- In jedem Fall werden spezifische Adapter benötigt, um die Daten auf Views zu mappen

```
public class UsersAdapter extends ArrayAdapter<User> {  
    public UsersAdapter(Context ctx, User[] users) { super(ctx, 0, users); }  
  
    @Override  
    public View getView(int position, View view, ViewGroup parent) {  
        1 User userItem = getItem(position);  
        2 if (view == null) {  
            view = LayoutInflater.from(getContext())  
                .inflate(R.layout.userview_layout, parent);  
        }  
        3 // TODO: populate fields/sub-views of view with data of userItem  
        return view;  
    }  
}  
  
public class UsersListActivity extends ListActivity {  
    @Override  
    protected void onCreate(final Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        final Users[] users = userDao().getAllUsers();  
        final UsersAdapter adapter = new UsersAdapter(this, users);  
        setListAdapter(adapter);  
    }  
}
```

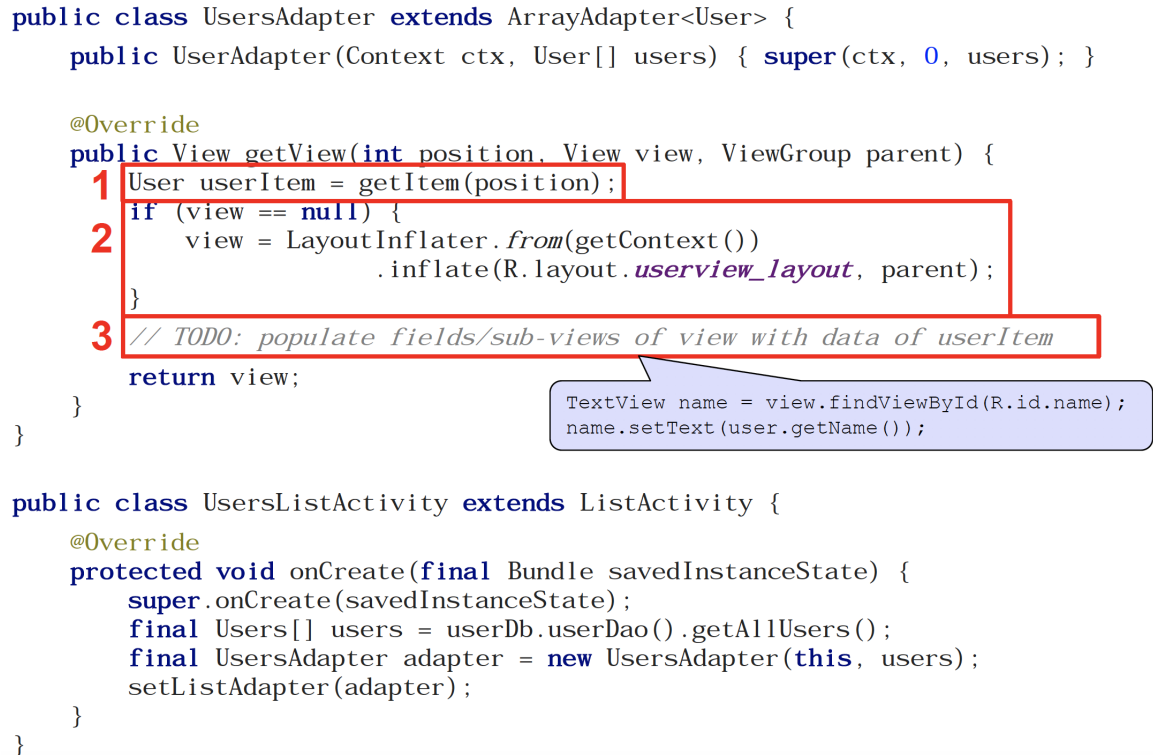


Abbildung 38: Codebeispiel für das Darstellen von DB-Einträgen in einer Liste

#### 3.5.1 Room - weitere Themen

### 3.6 Content Providers

- Content Provider stellen für andere Applikationen Daten bereit
- Die Daten stammen aus einer gekapselten DB **oder** aus dem privaten Dateisystem **oder** werden on-the-fly erzeugt
- Zugriff auf die Daten über URI (Uniform Ressource ID), Beispiel siehe in Abbildung 39
- Zwei Arten von URIs
  - Pfad (Bezeichnete Datenmenge, vgl. Verzeichnit mit Daten)
  - Item (Einzelnes Datenelement, vgl. einzelne Datei)

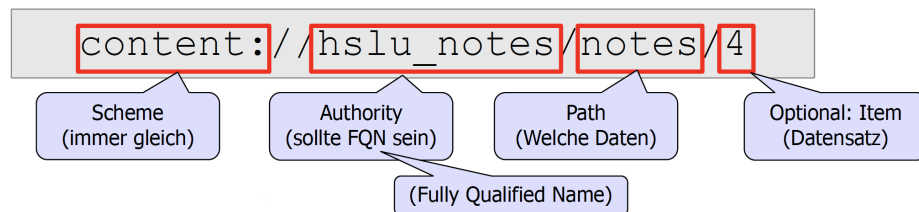


Abbildung 39: Aufbau eines URI

#### 3.6.1 Standard Content Providers

- Im Android-System gibt es bereits einige Content Providers, die genutzt werden können
  - Kontakte: Namen, Telefon-Nummern, Emails, Adressen, etc.
  - SMS/MMS: Erhaltene/Gesendete/Drafts SMS/MMS
  - Media Store: Auf Gerät gespeicherte Audio-, Video-, Bilder-Daten
  - Settings: Einstellungen für das Gerät
  - Kalender: Kalender, Events, Erinnerungen, Teilnehmer, etc.
- Daten sind meist in mehreren Tabellen abgelegt

### 3.7 Exkurs : REST-ful Webservices

- Webservices auf der Basis von HTTP
- Grundidee (in purer Form)
  - URL einer Ressourcensammlung (*<http://directory.com/contacts>*) oder URL einzelner Ressource (*<http://directory.com/contacts/17>*)
  - HTTP-Methode = Operation auf Daten (GET, PUT, POST, DELETE)
  - Antwort-Datenformat = XML, JSON, ...

Resource	GET	PUT	POST	DELETE
Collection URI, such <a href="http://directory.com/contacts/">http://directory.com/contacts/</a>	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URL is usually returned by the operation.	Delete the entire collection.
Element URI, such as <a href="http://directory.com/contacts/17">http://directory.com/contacts/17</a>	Retrieve a representation of the addressed collection member, expressed in an appropriate media type.	Replace the addressed member of the collection, or if it doesn't exist, create it.	Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Abbildung 40: Beispiel - Representational State Transfer

### 3.8 Content Resolver & Content Provider

- Zugriff auf einen Content Provider erfolgt über einen **Content Resolver**  
`Context.getContentResolver()`
  - Bietet DB-Methoden und Zugriff auf Content via Streams
    - \* CRUD: `insert()` / `query()` / `update()` / `delete()`
    - \* `openInputStream(uri)` / `openOutputStream(uri)`
  - Ein Content Resolver ist ein Proxy, der...
    - \* ...URI auflöst und zuständigen Content Provider sucht / findet
    - \* ...Interprozess-Kommunikation behandelt (aufrufende App ist meist in einem anderen Package als der aufgerufene Content Provider)
- Unter Umständen müssen die Permissions noch gesetzt werden (im Manifest)

#### 3.8.1 Zugriff auf Daten über Content Resolver & Query

```

1 Cursor cursor = getContentResolver().query(
2     contentUri,        // The content URI of the table
3     projection,        // The columns to return for each row
4     selectionClause,    // Selection criteria
5     selectionArgs,      // Selection criteria
6     sortOrder);        // The sort order for the returned row

```

Content Provider Query	SQL SELECT Query	Notes
contentUri	FROM table_name	contentUri maps to the table in the provider named table_name.
projection	Col, col, col,...	projection is an array of columns that should be included for each row retrieved.
selection	WHERE col = value	selection specifies the criteria for selecting rows.
selectionArgs	(No exact equivalent. Selection arguments replace ? placeholders in the selection clause.)	-
sortOrder	ORDER BY col,col,...	sortOrder specifies the order in which rows appear in the returned Cursor.

Abbildung 41: Vergleich: ContentProvider Query und SQL Query Parameter

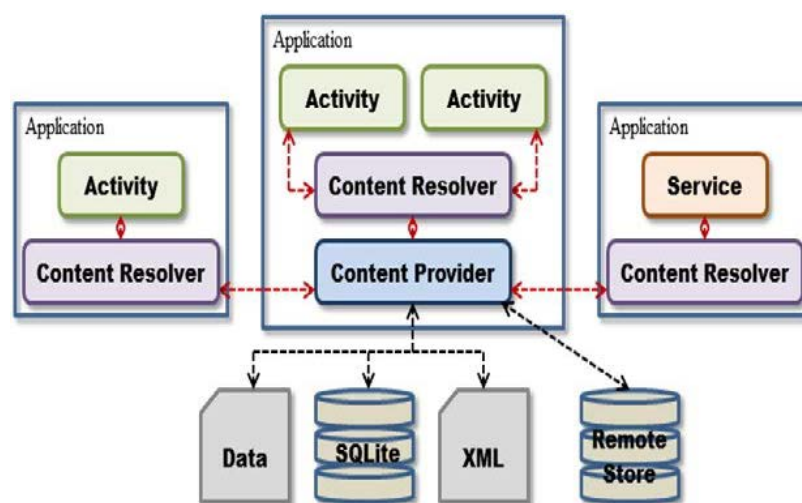


Abbildung 42: Content Provider - Anwendung (Data: Dateisystem, XML: Preferences)

- SMS des Systems sind über den Content Provider zugänglich  
(Benötigt Permission für SMS, diese testen und ggf. beantragen):
  - android.provider.Telephony.Sms  
(„Sub-Providers“ für Sent, Inbox, Draft, etc.)
  - Im Package android.provider.\* finden wir  
„Contract Klasse“ Telephone.Sms mit  
Hilfsklassen BaseColumns und Telephony.TextBasedSmsColumns  
(Hier findet man Content-URI und Spalten-Namen für Projections)

```
public void showSmsList(final View view) {
    final Cursor cursor = getContentResolver().query(
        Telephony.Sms.Inbox.CONTENT_URI, // content uri
        new String[]{ Telephony.Sms.Inbox._ID, Telephony.Sms.Inbox.BODY }, // projection
        null, null, null); // selection, selection args, sort order

    new AlertDialog.Builder(this)
        .setTitle("SMS in Inbox")
        .setCursor(cursor, null, Telephony.TextBasedSmsColumns.BODY)
        .setNeutralButton("Ok", null)
        .create()
        .show();
}
```

Abbildung 43: Anwendungsbeispiel - Alle SMS mit Text anzeigen

Jeder Content Provider bietet eine eigene Standard-API, in der Android Dokumentation sind die Zugriffe auf Kontakte und Kalender gut dokumentiert (da dies eher komplizierte Modelle sind). Einstiegspunkt für die meisten Provider: android.provider.\*

### 3.9 Eigener Content Provider

- Um eigenen Content Provider zu schreiben, muss die eigene Klasse von der abstrakten Klasse android.content.ContentProvider ableiten
- Wird bei App-Start hochgefahren und bleibt aktiv, in onCreate() kann eine Initialisierung vorgenommen werden (einzige Lifecycle-Methode)
- CRUD-Methoden: query, insert, update, delete (muss nicht alle implementieren)  
(Möglichkeit, einen read-only Content Provider anzulegen)

#### Demo: Content Provider für Notizen

- Dialog zeigt Notizen an
- Nur für internen Gebrauch
  - exported=false
- NotesProvider: Konstanten definiert in NotesContract
- Aufrufender Code in Activity:

```
public void readNotesFromContentProviderOnClick(final View view) {
    final Cursor cursor = getContentResolver().query(
        Uri.parse(NotesContract.CONTENT_URI), NotesContract.PROJECTION,
        null, // SELECT *
        null, NotesContract.COLUMN_TITLE);

    new AlertDialog.Builder(this).setTitle("Notizen via ContentProvider")
        .setCursor(cursor, null, NotesContract.COLUMN_TITLE)
        .setPositiveButton("Toll!", null)
        .setNegativeButton("Abbruch", null).create().show();
}
```

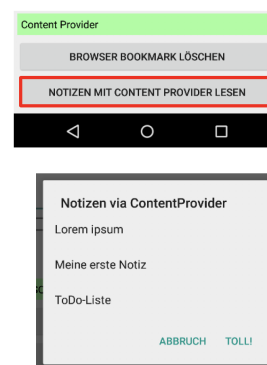


Abbildung 44: Anwendungsbeispiel - Content Provider für Notizen